pusion Documentation

Admir Obralija, Yannick Wilhelm

Dec 13, 2022

CONTENTS

Pusion (Python Universal Fusion) is a flexible framework for combining multi-classifier decisions in Python.

The framework accommodates a variety of fusion methods adapted from ensemble techniques and pattern recognition. The general purpose is to improve the classification performance over the input classifiers and to determine the compatibility of individual fusion methods to the given problem. Pusion handles an unlimited number of classifiers and also tolerates different forms of classification outputs. This includes the multiclass and multilabel classification, as well as crisp and continuous class assignments.

The framework is originally designed for combining fault detection and diagnosis methods. However, its application is not limited to this area.

This documentation includes the following sections:

CHAPTER

ONE

OVERVIEW

1.1 Introduction

In general, there exist two main approaches to combine multiple classifiers in order to obtain reasonable classification outputs for unseen samples: the classifier selection and the classifier fusion. The latter one is followed by the presented framework and illustrated in Fig. 1.1.

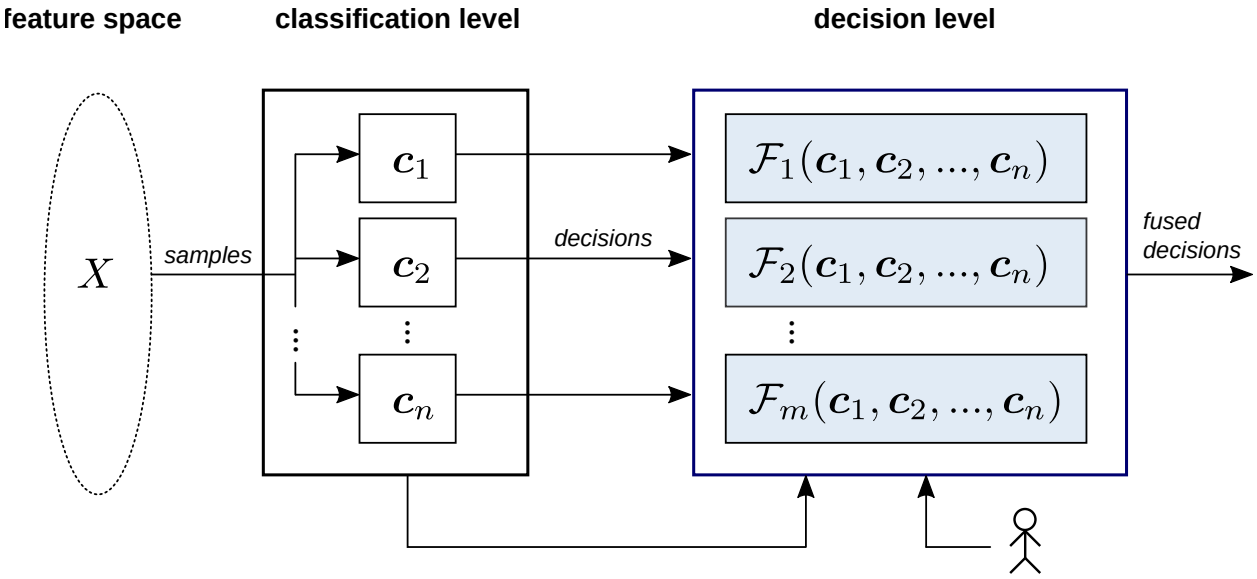

Fig. 1.1: Architectural embedding of the decision fusion framework

However, there are significant properties arising from such classifier ensembles as parameters which need to be considered in the whole decision fusion process. These parameters are forming a configuration which consists of:

- Decision fusion method (*combiner*)
- Classification problem
- Class assignment type
- Classification coverage

The *combiner* states an explicit method provided by the framework which should be applied on the input data set. Available core methods are listed in the following section.

The *classification problem* refers either to a multiclass or to a multilabel classification problem. In the multiclass case, a sample is always classified into one class, while in the multilabel case, more than one class may be assigned to a sample.

Pusion operates on classification data which is given by class assignments. The class assignment describes memberships to each individual class for a sample. A *class assignment type* is either crisp or continuous. Crisp assignments are equivalent to labels and continuous assignments represent probabilities for each class being true.

The *classification coverage* states for each input classifier, which classes it is able to decide. A classifier ensemble may yield a redundant, complementary or complementary-redundant coverage.

1.2 Core methods

The following core decision fusion methods are supported by *pusion* and classified according to the evidence resolution they accept. With lower evidence resolution, a weaker a-priori information about individual classifiers can be taken into account during the fusion process. Utility-based methods do not take any further information about classifiers into account. In cases where no evidence is available for a certain classification data set, a utility-based method is a reasonable choice. Evidence-based methods are recommended in cases where evidence (e.g. confusion matrices) but no training data is available for each classifier. Trainable methods provide the highest evidence resolution, since decision outputs are required from the ensemble for each sample during the training phase. Therefore, each *trainable combiner* is able to calculate any kind of evidence based on the training data and is even able to analyse the behaviour of each classification method from the ensemble.

Utility-based methods (low evidence resolution):

- Borda Count (*BC*)
- Cosine Similarity (*COS*)
- Macro Majority Vote (*MAMV*)
- Micro Majority Vote (*MIMV*)
- Simple Average (*AVG*)

Evidence-based methods (medium evidence resolution):

- Naive Bayes (*NB*)
- Weighted Voting (*WV*)

Trainable methods (highest evidence resolution):

- Behaviour Knowledge Space (*BKS*)
- Decision Templates (*DT*)
- k Nearest Neighbors (*KNN*)
- Dempster Shafer (*DS*)
- Maximum Likelihood (*MLE*)
- Neural Network (*NN*)

1.3 Data input and output

The input type used for classification data is generic and applies to all provided decision fusion methods. It is given by a 3D numpy.ndarray tensor, which is illustrated in Fig. 1.2.

Fig. 1.2: Illustration of the input tensor for a multilabel problem with crisp assignments (3 samples, 4 classes, 2 classifiers).

The same applies also to the pusion's return, except that the output matrix is a 2D *numpy.ndarray*.

Note: In case of complementary-redundant decisions, the coverage needs to be specified besides ordinary python lists, which are used as an alternative to the *numpy.ndarray*.

1.4 AutoFusion

The framework provides an additional fusion method *AutoCombiner* which is able to the detect the configuration based on the input classification data and to automatically select the fusion method with the best classification performance for the given problem. The *AutoCombiner* bundles all methods provided by the framework and probes each of them for the application on the given classification data. The *AutoCombiner* is transparent to the user as each of the core fusion methods.

1.5 Generic fusion

In contrast to the *AutoCombiner*, the *GenericCombiner* retrieves fusion results obtained by all compatible core methods by means of a *numpy.ndarray* tensor. In this case, the evaluation as well as the method selection is handed over to the user.

1.6 Further functionalities

- Classification data and coverage generation (see module *generator*)
- Useful transformations for decision outputs, e.g. multilabel to multiclass conversion (see module *transformer*)
- Evaluation methods for different classification and coverage types (see class *Evaluation*)

CHAPTER

TWO

INSTALL PUSION

2.1 General requirements

- Python >= 3.6

2.2 Package requirements

- numpy >= 1.20.2
- scipy >= 1.6.2
- scikit-learn >= 0.24.1
- setuptools >= 54.2.0
- pandas >= 1.2.3
- matplotlib >= 3.4.1

2.3 Preparation

To generate the python distribution archives of *pusion*, update the PyPA's build to the latest version. Under *Windows* run in your python environment the following command:

```
py -m pip install --upgrade build
```

If you are using *MacOS* or *Unix*, run in your python environment:

```
python3 -m pip install --upgrade build
```

After cloning *pusion* from GitHub to your local computer, enter the *pusion* directory where the file *pyproject.toml* is located. Under this directory run the following *build* command. For *Windows* users:

```
py -m build
```

For *MacOS* or *Unix* user:

```
python3 -m build
```

Once successfully executed, two files are generated in the *dist* subfolder within the project's root folder. The *tar.gz* file is the source distribution and the *whl* file is the built distribution.

2.4 Installation

The generated wheel can be installed using the `pip3` command, which also installs all required packages for *pusion*.

```
pip3 install dist/pusion-<version>-py3-none-any.whl
```

CHAPTER

THREE

API REFERENCE

3.1 pusion.auto package

3.1.1 pusion.auto.auto_combiner

class pusion.auto.auto_combiner.**AutoCombiner**

Bases: *pusion.auto.generic_combiner.GenericCombiner*

The *AutoCombiner* allows for automatic decision fusion using all methods provided by the framework, which are applicable to the given problem. The key feature of this combiner is the transparency in terms of it's outer behaviour. Based on the usage (i.e. method calls) and the automatically detected configuration, the *AutoCombiner* preselects all compatible methods from *pusion.core*. The main purpose is to retrieve fusion results obtained by the methods with the best performance without further user interaction.

train(*decision_tensor*, *true_assignments*, ***kwargs*)

Train the AutoCombiner (AC) model. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs using the AutoCombiner (AC) model. Both continuous and crisp classification outputs are supported. This procedure involves selecting the best method regarding its classification performance in case of a trained AC.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

set_validation_size(*validation_size*)
Set the validation size, based on which the training data is split and the best combiner is selected.

Parameters **validation_size** – A *float* between *0* and *1.0*. Ratio of the validation data set.

combine_par(*decision_tensor*)
Combine decision outputs by GC. Both continuous and crisp classification outputs are supported. This procedure involves combining decision outputs by each individual method which is applicable to the detected configuration. Each combine procedure is spawned in a separate thread and thus performed in parallel.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

combine_seq(*decision_tensor*)
Combine decision outputs by GC. Both continuous and crisp classification outputs are supported. This procedure involves combining decision outputs by each individual method which is applicable to the detected configuration. Each combine procedure is initiated in sequence.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

get_combiner_type_selection()

Returns list of combiner types established by usage.

get_combiners()

Returns list of core methods preselected by the *AutoCombiner*.

get_multi_combiner_decision_tensor()

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

get_multi_combiner_runtimes()

Returns A *tuple* of two lists of tuples describing the train and combine runtimes respectively. Each inner tuple key value indexes the list of preselected fusion methods (retrievable by `get_combiners()`).

get_pac()

Returns *tuple* of detected problem, assignment type and coverage type.

`get_selected_combiner()`

Returns The method selected by the *AutoCombiner*.

`classmethod obtain(config)`
Obtain a combiner registered by the framework.

Parameters **config** – *pusion.model.configuration.Configuration*. User-defined configuration.

Returns A *combiner* object.

`set_coverage(coverage)`
Set the coverage in case of complementary-redundant classification data.

Parameters **coverage** – The coverage is described by using a nested list. Each list describes the classifier based on its position. Elements of those lists (integers) describe the actual class coverage of the respective classifier. E.g., with `[[0,1], [0,2,3]]` the classes 0,1 are covered by the first classifier and 0,2,3 are covered by the second one.

`set_evidence(evidence)`
Set the evidence for evidence based combiners. This method preselects all combiners of type *EvidenceBasedCombiner*.

Parameters **evidence** – *numpy.array* of shape *(n_classifiers, n_classes, n_classes)*. Confusion matrices for each of *n* classifiers.

`set_parallel(parallel=True)`
Set whether the training and the combining of selected combiners should be executed sequentially or in parallel. :param parallel: If *True*, training and combining is performed in parallel respectively. Otherwise in sequence.

`train_par(decision_tensor, true_assignments)`
Train the Generic Combiner by training individual combiners in parallel. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

`train_seq(decision_tensor, true_assignments)`
Train the Generic Combiner by training individual combiners in sequence. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

get_eval_metric()

Returns The metric used for the selection of the best performing combiner.

3.1.2 pusion.auto.generic_combiner

class pusion.auto.generic_combiner.**GenericCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*, *pusion.core.combiner.EvidenceBasedCombiner*, *pusion.core.combiner.UtilityBasedCombiner*

The *GenericCombiner* (GC) allows for automatic decision fusion using all methods provided by the framework, which are applicable to the given problem. The key feature of this combiner is the transparency in terms of it's outer behaviour. Based on the usage (i.e. method calls) and the automatically detected configuration, the *GenericCombiner* preselects all compatible methods from *pusion.core*. The main purpose is to retrieve fusion results obtained by the all applicable methods. The main difference to the *AutoCombiner* is that decision fusion results are handed over to the user for further comparison and selection. Thus, GC is not suitable for the online fusion.

set_coverage(*coverage*)

Set the coverage in case of complementary-redundant classification data.

Parameters **coverage** – The coverage is described by using a nested list. Each list describes the classifier based on its position. Elements of those lists (integers) describe the actual class coverage of the respective classifier. E.g., with `[[0,1], [0,2,3]]` the classes 0,1 are covered by the first classifier and 0,2,3 are covered by the second one.

set_evidence(*evidence*)

Set the evidence for evidence based combiners. This method preselects all combiners of type *EvidenceBasedCombiner*.

Parameters **evidence** – *numpy.array* of shape *(n_classifiers, n_classes, n_classes)*. Confusion matrices for each of *n* classifiers.

train(*decision_tensor*, *true_assignments*)

Train the Generic Combiner. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

train_par(*decision_tensor*, *true_assignments*)

Train the Generic Combiner by training individual combiners in parallel. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

train_seq(*decision_tensor*, *true_assignments*)

Train the Generic Combiner by training individual combiners in sequence. This method detects the configuration based on the `decision_tensor` and trains all trainable combiners that are applicable to this configuration.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs using the AutoCombiner (AC) model. Both continuous and crisp classification outputs are supported. This procedure involves combining decision outputs by each individual method which is applicable to the detected configuration.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

combine_par(*decision_tensor*)

Combine decision outputs by GC. Both continuous and crisp classification outputs are supported. This procedure involves combining decision outputs by each individual method which is applicable to the detected configuration. Each combine procedure is spawned in a separate thread and thus performed in parallel.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

combine_seq(*decision_tensor*)

Combine decision outputs by GC. Both continuous and crisp classification outputs are supported. This procedure involves combining decision outputs by each individual method which is applicable to the detected configuration. Each combine procedure is initiated in sequence.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

get_pac()

Returns *tuple* of detected problem, assignment type and coverage type.

get_combiners()

Returns list of core methods preselected by the *AutoCombiner*.

get_combiner_type_selection()

Returns list of combiner types established by usage.

get_multi_combiner_decision_tensor()

Returns list of *numpy.array* of shape *(n_samples, n_classifiers)*. Fusion results obtained by selected fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

get_multi_combiner_runtimes()

Returns A *tuple* of two lists of tuples describing the train and combine runtimes respectively. Each inner tuple key value indexes the list of preselected fusion methods (retrievable by `get_combiners()`).

set_parallel(*parallel=True*)

Set whether the training and the combining of selected combiners should be executed sequentially or in parallel. :param parallel: If *True*, training and combining is performed in parallel respectively. Otherwise in sequence.

3.1.3 pusion.auto.detector module

pusion.auto.detector.**determine_problem**(*decision_outputs*)

Determine the classification problem based on the decision outputs.

Parameters **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Returns *string* constant *'MULTI_CLASS'* or *'MULTI_LABEL'*. See *pusion.util.constants.Problem*.

pusion.auto.detector.**determine_assignment_type**(*decision_outputs*)

Determine the assignment type based on the decision outputs.

Parameters **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Returns *string* constant *'CRISP'* or *'CONTINUOUS'*. See *pusion.util.constants.AssignmentType*.

pusion.auto.detector.**determine_coverage_type**(*coverage*)
Determine the coverage type.

Parameters **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

Returns *string* constant *'REDUNDANT'*, *'COMPLEMENTARY'* or *'COMPLEMENTARY_REDUNDANT'*. See *pusion.util.constants.CoverageType*.

pusion.auto.detector.**determine_pac**(*decision_outputs*, *coverage=None*)
Determine the PAC-tuple (problem, assignment type and coverage type) based on the given decision outputs and coverage.

Parameters

- **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

Returns *tuple* of string constants representing the PAC. See *pusion.util.constants.Problem*, *pusion.util.constants.AssignmentType* and *pusion.util.constants.CoverageType*.

3.2 pusion.control package

3.2.1 pusion.control.decision_processor module

class pusion.control.decision_processor.**DecisionProcessor**(*config:* pusion.model.configuration.Configuration)

Bases: `object`

DecisionProcessor is the main user interface of the decision fusion framework. It provides all methods for selecting combiners including the *AutoCombiner* and the *GenericCombiner*. It also ensures uniformity and correct use of all *pusion.core* combiners.

Parameters **config** – *pusion.model.configuration.Configuration*. User-defined configuration.

set_coverage(*coverage*)
Set the coverage in case of complementary-redundant classification data.

Parameters **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list. E.g., with `[[0,1], [0,2,3]]` the classes 0,1 are covered by the first classifier and 0,2,3 are covered by the second one.

set_evidence(*evidence*)
Set the evidence for evidence-based combiners. The evidence is given by confusion matrices calculated according to Kuncheva[1].

Parameters **evidence** – *list* of *numpy.array* elements of shape *(n_classes, n_classes)*. Confusion matrices for each ensemble classifier.

[1] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

`set_data_split_ratio`(*validation_size*)
Set the size of the validation data used by the AutoCombiner to evaluate all applicable fusion methods in order to select the combiner with the best classification performance. Accordingly, the other data of size *1-validation_size* is used to train all individual combiners.

Parameters `validation_size` – A *float* between *0* and *1.0*. Ratio of the validation data set.

`train`(*y_ensemble_valid*, *y_valid*, ***kwargs*)
Train the combiner model determined by the configuration.

> **Warning:** A trainable combiner is always trained with the validation dataset provided by ensemble classifiers.

Parameters

- `y_ensemble_valid` – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

  Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- `y_valid` – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.
- `**kwargs` – The ***kwargs* parameter may be used to use additional test data for the AutoFusion selection procedure.

`combine`(*y_ensemble_test*)
Combine decision outputs using the combiner model determined by the configuration.

Parameters `y_ensemble_test` – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classes)*. A matrix of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `y_ensemble_test` input tensor.

`get_multi_combiner_decision_output`()
Retrieve the decision fusion outputs obtained by multiple combiners. This function is only callable for configurations including *AutoCombiner* or *GenericCombiner* as a method.

Returns *list* of *numpy.array* elements of shape *(n_samples, n_classes)*. Fusion results obtained by multiple fusion methods. The list is aligned with the list of preselected fusion methods (retrievable by `get_combiners()`).

`get_optimal_combiner`(*eval_metric=None*)
Retrieve the combiner with the best classification performance obtained by the framework, i.e. the *AutoCombiner* or the *GenericCombiner*. In case of combining with the *GenericCombiner*, an *Evaluation* needs to be set by `set_evaluation`.

Returns The combiner object.

`get_combiners`()
Retrieve combiners (core methods) which are preselected by the framework according to the auto-detected configuration. :return: *list* of combiner objects obtained by the *GenericCombiner* or *AutoCombiner*.

get_combiner()

Returns Selected combiner object.

report(*eval_metric=None*)

Returns The textual evaluation report.

info()
Retrieve the information, the automatic combiner selection is based on.

Returns

tuple of the form *((A, B, C), D)*, whereby *A* represents the classification problem *('MULTI_CLASS' or 'MULTI_LABEL')*, *B* the assignment type *('CRISP' or 'CONTINUOUS')* and *C* the coverage type *('REDUNDANT', 'COMPLEMENTARY' or 'COMPLEMENTARY_REDUNDANT')*.

D contains the combiner type selection as a *list*. Possible combiner types are *UtilityBasedCombiner*, *TrainableCombiner* and *EvidenceBasedCombiner*.

get_multi_combiner_runtimes()
Retrieve the train and combine runtime for each combiner used during a generic fusion.

Returns A *tuple* of two lists of tuples describing the train and combine runtimes respectively. Each inner tuple key value indexes the list of preselected fusion methods (retrievable by `get_combiners()`).

set_evaluation(*evaluation*)

Parameters **evaluation** – `pusion.control.evaluation.Evaluation` object, a combiner evaluation was performed with.

set_parallel(*parallel=True*)
Set whether the training and the combining of selected combiners should be executed sequentially or in parallel. :param parallel: If *True*, training and combining is performed in parallel respectively. Otherwise in sequence.

3.3 pusion.core package

3.3.1 pusion.core.behaviour_knowledge_space_combiner

class pusion.core.behaviour_knowledge_space_combiner.**BehaviourKnowledgeSpaceCombiner**
Bases: *pusion.core.combiner.TrainableCombiner*

The *BehaviourKnowledgeSpaceCombiner* (BKS) is adopted from the decision fusion method originally proposed by Huang, Suen et al.[1]. BKS analyses the behaviour of multiple classifiers based on their classification outputs with respect to each available class. This behaviour is recorded by means of a lookup table, which is used for final combination of multiple classification outputs for a sample.

[1] Yea S Huang and Ching Y Suen. The behavior-knowledge space method for combination of multiple classifiers. In *IEEE computer society conference on computer vision and pattern recognition*, 347–347. Institute of Electrical Engineers Inc (IEEE), 1993.

train(*decision_tensor*, *true_assignments*)

Train the Behaviour Knowledge Space model (BKS) by extracting the classification configuration from all classifiers and summarizing samples of each true class that leads to that configuration. This relationship is recorded in a lookup table. Only crisp classification outputs are supported.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by the Behaviour Knowledge Space (BKS) method. This procedure involves looking up the most representative class for a given classification output regarding the behaviour of all classifiers in the ensemble. Only crisp classification outputs are supported. If a trained lookup entry is not present for a certain classification configuration, no decision fusion can be made for the sample, which led to that configuration. In this case, the decision fusion is a zero vector.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which are obtained by the best representative class for a certain classifier's behaviour per sample. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.behaviour_knowledge_space_combiner.**CRBehaviourKnowledgeSpaceCombiner**

Bases: *pusion.core.behaviour_knowledge_space_combiner.BehaviourKnowledgeSpaceCombiner*

The *CRBehaviourKnowledgeSpaceCombiner* is a modification of *BehaviourKnowledgeSpaceCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)

Train the Behaviour Knowledge Space model (BKS) by extracting the classification configuration from all classifiers and summarizing samples of each true class that leads to that configuration. This relationship is recorded in a lookup table. Only crisp classification outputs are supported.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which is considered true for each sample during the training procedure.

combine(*decision_outputs*)

Combine decision outputs by the Behaviour Knowledge Space (BKS) method. This procedure involves looking up the most representative class for a given classification output regarding the behaviour of all classifiers in the ensemble. Only crisp classification outputs are supported. If a trained lookup entry is not present for a certain classification configuration, no decision fusion can be made for the sample, which led to that configuration. In this case, the decision fusion is a zero vector.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which are obtained by the best representative class for a certain classifier's behaviour per sample. Axis 0 represents samples and axis 1 all the class labels which are provided by the coverage.

3.3.2 pusion.core.borda_count_combiner

class pusion.core.borda_count_combiner.**BordaCountCombiner**

Bases: *pusion.core.combiner.UtilityBasedCombiner*

The *BordaCountCombiner* (BC) is a decision fusion method that establishes a ranking between label assignments for a sample. This ranking is implicitly given by continuous support outputs and is mapped to different amounts of votes (0 of L votes for the lowest support, and $L - 1$ votes for the highest one). A class with the highest sum of these votes (borda counts) across all classifiers is considered as a winner for the final decision.

combine(*decision_tensor*)

Combine decision outputs by the Borda Count (BC) method. Firstly, the continuous classification is mapped to a ranking with respect to available classes for each sample. Those rankings are then summed up across all classifiers to establish total votes (borda counts) for each class in a sample. The class with the highest number of borda counts is considered as decision fusion. Only continuous classification outputs are supported.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.borda_count_combiner.**CRBordaCountCombiner**

Bases: *pusion.core.borda_count_combiner.BordaCountCombiner*

The *CRBordaCountCombiner* is a modification of *BordaCountCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as *0*, respectively. To call *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

combine(*decision_outputs*)

Combine complementary-redundant decision outputs by the Borda Count (BC) method. Firstly, the continuous classification is mapped to a ranking with respect to available classes for each sample. Those rankings are then summed up across all classifiers to establish total votes (borda counts) for each class in a sample. The class with the highest number of borda counts is considered as decision fusion. Only continuous classification outputs are supported.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.3 pusion.core.combiner

class pusion.core.combiner.**Combiner**

Bases: object

Combiner's root class. This class as well as the subclasses in this module set the structure for each combiner provided by the framework. It also accommodates methods and attributes which are essential for combiner's registration and obtainment.

Each combiner to be registered in the framework, needs to base at least one of the following classes:

- *UtilityBasedCombiner*
- *TrainableCombiner*
- *EvidenceBasedCombiner*.

Furthermore, it needs a *list* _SUPPORTED_PAC of supported PAC tuples set as a class attribute. A PAC tuple is a *tuple* of string constants (classification problem, assignment type and coverage type). See:

- *pusion.util.constants.Problem*
- *pusion.util.constants.AssignmentType*
- *pusion.util.constants.CoverageType*

respectively. Example of a new combiner:

```
class NewCombiner(TrainableCombiner):

_SUPPORTED_PAC = [
    (Problem.MULTI_CLASS, AssignmentType.CRISP, CoverageType.REDUNDANT),
    (Problem.MULTI_CLASS, AssignmentType.CONTINUOUS, CoverageType.REDUNDANT)
]

def train(self, decision_outputs, true_assignments):
    pass

def combine(self, decision_outputs):
    pass
```

Warning: Note that a new combiner also needs to be inserted into the `pusion.Method` class within *pusion.__init__.py* file.

classmethod obtain(*config*)

Obtain a combiner registered by the framework.

Parameters **config** – *pusion.model.configuration.Configuration*. User-defined configuration.

Returns A *combiner* object.

abstract combine(*decision_tensor*)

Abstract method. Combine decision outputs by combiner's implementation.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classes)*. A matrix of class assignments which represents fused decisions obtained by combiner's implementation. Axis 0 represents samples

and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

set_coverage(*coverage*)
Set the coverage for complementary-redundant decisions.

Parameters **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

class pusion.core.combiner.**UtilityBasedCombiner**
Bases: *pusion.core.combiner.Combiner*

A combiner of type *UtilityBasedCombiner* fuses decisions solely based on the outputs of the ensemble classifiers. It does not take any further information or evidence about respective ensemble classifiers into account.

abstract combine(*decision_tensor*)
Abstract method. Combine decision outputs by combiner's implementation.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classes)*. A matrix of class assignments which represents fused decisions obtained by combiner's implementation. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.combiner.**TrainableCombiner**
Bases: *pusion.core.combiner.Combiner*

A combiner of type *TrainableCombiner* needs to be trained using decision outputs of the ensemble classifiers with true class assignments in order to combine decisions of unknown samples.

abstract combine(*decision_tensor*)
Abstract method. Combine decision outputs by combiner's implementation.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classes)*. A matrix of class assignments which represents fused decisions obtained by combiner's implementation. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

abstract train(*decision_tensor*, *true_assignments*, ***kwargs*)
Abstract method. Train combiner's implementation using decision outputs an appropriate true assignments.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of class assignments which are considered true for each sample during the training procedure.
- ****kwargs** – The ***kwargs* parameter may be used to use additional test data for the AutoFusion selection procedure.

class pusion.core.combiner.**EvidenceBasedCombiner**
Bases: *pusion.core.combiner.Combiner*

A combiner of type *EvidenceBasedCombiner* takes an additional evidence into account while combining outputs of ensemble classifiers. Thus, it is able to empower better classifiers in order to obtain a fusion result with higher overall classification performance.

abstract combine(*decision_tensor*)
Abstract method. Combine decision outputs by combiner's implementation.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classes)*. A matrix of class assignments which represents fused decisions obtained by combiner's implementation. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

abstract set_evidence(*evidence*)
Abstract method. Set the evidence for evidence-based combiner implementations. The evidence is given by confusion matrices calculated according to Kuncheva[1].

Parameters **evidence** – *list* of *numpy.array* elements of shape *(n_classes, n_classes)*. Confusion matrices for each ensemble classifier.

3.3.4 pusion.core.cosine_similarity_combiner

class pusion.core.cosine_similarity_combiner.**CosineSimilarityCombiner**
Bases: *pusion.core.combiner.UtilityBasedCombiner*

The *CosineSimilarityCombiner* considers the classification assignments to ℓ classes as vectors from an ℓ-dimensional vector space. The normalized cosine-similarity measure between two vectors x and y is calculated as

$$cos(x, y) = \frac{x \cdot y}{|x||y|} .$$

The cosine-similarity is calculated pairwise and accumulated for each classifier for one specific sample. The fusion is represented by a classifier which shows the most similar classification output to the output of all competing classifiers.

combine(*decision_tensor*)
Combine decision outputs with as an output that accommodates the highest cosine-similarity to the output of all competing classifiers. In other words, the best representative classification output among the others is selected according to the highest cumulative cosine-similarity. This method supports both, continuous and crisp classification outputs.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by the highest cumulative cosine-similarity. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.cosine_similarity_combiner.**CRCosineSimilarity**
Bases: *pusion.core.cosine_similarity_combiner.CosineSimilarityCombiner*

[1] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

The *CRCosineSimilarity* is a modification of *CosineSimilarityCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as *0*, respectively. To call *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

combine(*decision_outputs*)

Combine complementary-redundant decision outputs with as an output that accommodates the highest cosine-similarity to the output of all competing classifiers. In other words, the best representative classification output among the others is selected according to the highest cumulative cosine-similarity. This method supports both, continuous and crisp classification outputs.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.5 pusion.core.decision_templates_combiner

class pusion.core.decision_templates_combiner.**DecisionTemplatesCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*

The *DecisionTemplatesCombiner* (DT) is adopted from the decision fusion method originally proposed by Kuncheva[1]. A decision template is the average matrix of all decision profiles, which correspond to samples of one specific class. A decision profile contains classification outputs from all classifiers for a sample in a row-wise fashion. The decision fusion is performed based on distance calculations between decision templates and the decision profile generated from the ensemble outputs.

train(*decision_tensor*, *true_assignments*)

Train the Decision Templates Combiner model by precalculating decision templates from given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure involves calculating means of decision profiles (decision templates) for each true class.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by using the Decision Templates method. Both continuous and crisp classification outputs are supported. Combining requires a trained *DecisionTemplatesCombiner*.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by the minimum distance between decision profiles of `decision_tensor` and precalculated decision templates. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

[1] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

class pusion.core.decision_templates_combiner.**CRDecisionTemplatesCombiner**

Bases: *pusion.core.decision_templates_combiner.DecisionTemplatesCombiner*

The *CRDecisionTemplatesCombiner* is a modification of *DecisionTemplatesCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)

Train the Decision Templates Combiner model by precalculating decision templates from given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure involves calculating means of decision profiles (decision templates) for each true class.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)

Combine decision outputs by using the Decision Templates method. Both continuous and crisp classification outputs are supported. Combining requires a trained *CRDecisionTemplatesCombiner*.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.6 pusion.core.k_nearest_neighbors_combiner.py

class pusion.core.k_nearest_neighbors_combiner.**KNNCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*

The *KNNCombiner* (kNN) is a learning and classifier-based combiner that converts multiple decision outputs into new features, which in turn are used to train this combiner. The kNN combiner (k=5) uses uniform weights for all neighbors and the standard Euclidean metric for the distance.

train(*decision_tensor*, *true_assignments*)

Train the kNN combiner by fitting the *k* nearest neighbors (k=5) model with given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by the *k* nearest neighbors (k=5) model. Both continuous and crisp classification outputs are supported. Combining requires a trained `DecisionTreeCombiner`. This procedure transforms decision outputs into a new feature space.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by kNN. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.k_nearest_neighbors_combiner.**CRKNNCombiner**

Bases: *pusion.core.k_nearest_neighbors_combiner.KNNCombiner*

The *CRKNNCombiner* is a modification of *KNNCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)

Train the kNN combiner model by fitting the *k* nearest neighbors (k=5) model with given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)

Combine decision outputs by the *k* nearest neighbors (k=5) model. Both continuous and crisp classification outputs are supported. Combining requires a trained `DecisionTreeCombiner`. This procedure transforms decision outputs into a new feature space.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.7 pusion.core.dempster_shafer_combiner

class pusion.core.dempster_shafer_combiner.**DempsterShaferCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*

The *DempsterShaferCombiner* (DS) fuses decision outputs by means of the Dempster Shafer evidence theory referenced by Polikar[1] and Ghosh et al.[2]. DS involves computing the *proximity* and *belief* values per classifier and class, depending on a sample. Then, the total class support is calculated using the Dempster's rule as the product of belief values across all classifiers to each class, respectively. The class with the highest product is considered as a fused decision. DS shares the same training procedure with the `DecisionTemplatesCombiner`.

train(*decision_tensor*, *true_assignments*)

Train the Dempster Shafer Combiner model by precalculating decision templates from given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure involves calculations mean decision profiles (decision templates) for each true class assignment.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by using the Dempster Shafer method. Both continuous and crisp classification outputs are supported. Combining requires a trained *DempsterShaferCombiner*. This procedure involves computing the proximity, the belief values, and the total class support using the Dempster's rule.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by the maximum class support. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.dempster_shafer_combiner.**CRDempsterShaferCombiner**

Bases: *pusion.core.dempster_shafer_combiner.DempsterShaferCombiner*

The *CRDempsterShaferCombiner* is a modification of *DempsterShaferCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)

Train the Dempster Shafer Combiner model by precalculating decision templates from given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure involves calculations mean decision profiles (decision templates) for each true class assignment.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.

[1] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6(3):21–45, 2006.

[2] Kaushik Ghosh, Yew Seng Ng, and Rajagopalan Srinivasan. Evaluation of decision fusion strategies for effective collaboration among heterogeneous fault diagnostic methods. *Computers & chemical engineering*, 35(2):342–355, 2011.

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)

Combine decision outputs by using the Dempster Shafer method. Both continuous and crisp classification outputs are supported. Combining requires a trained *DempsterShaferCombiner*. This procedure involves computing the proximity, the belief values, and the total class support using the Dempster's rule.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp or continuous class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.8 pusion.core.macro_majority_vote_combiner

class pusion.core.macro_majority_vote_combiner.**MacroMajorityVoteCombiner**

Bases: *pusion.core.combiner.UtilityBasedCombiner*

The *MacroMajorityVoteCombiner* (MAMV) is based on a variation of the general majority vote method. The fusion consists of a decision vector which is given by the majority of the classifiers in the ensemble for a sample. MAMV does not consider outputs for each individual class (macro).

combine(*decision_tensor*)

Combine decision outputs by majority voting across all classifiers considering the most common classification assignment (macro). Only crisp classification outputs are supported.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments obtained by MAMV. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.9 pusion.core.maximum_likelihood_combiner

class pusion.core.maximum_likelihood_combiner.**MaximumLikelihoodCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*

The *MaximumLikelihoodCombiner* (MLE) is a combiner that estimates the parameters μ (sample means) and σ (sample variances) of the Gaussian probability density function for each class ω. Multiple decision outputs for a sample are converted into a new feature space.

The fusion is performed by evaluating the class conditional density

$$p(x|\omega) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right).$$

of a transformed sample x for each available class ω, respectively. The class with the highest likelihood is considered as winner and thus forms the decision fusion.

train(*decision_tensor*, *true_assignments*)
Train the Maximum Likelihood combiner model by calculating the parameters of gaussian normal distribution (i.e. means and variances) from the given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)
Combine decision outputs by the Maximum Likelihood method. This procedure involves evaluating the class conditional density as described above. Both continuous and crisp classification outputs are supported. Combining requires a trained *MaximumLikelihoodCombiner*.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by MLE. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.maximum_likelihood_combiner.**CRMaximumLikelihoodCombiner**
Bases: *pusion.core.maximum_likelihood_combiner.MaximumLikelihoodCombiner*

The *CRMaximumLikelihoodCombiner* is a modification of *MaximumLikelihoodCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)
Train the Maximum Likelihood combiner model by calculating the parameters of gaussian normal distribution (i.e. means and variances) from the given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)
Combine decision outputs by the Maximum Likelihood method. This procedure involves evaluating the class conditional density as described above. Both continuous and crisp classification outputs are supported. Combining requires a trained *MaximumLikelihoodCombiner*.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by MLE. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.10 pusion.core.micro_majority_vote_combiner

class pusion.core.micro_majority_vote_combiner.**MicroMajorityVoteCombiner**

Bases: *pusion.core.combiner.UtilityBasedCombiner*

The *MicroMajorityVoteCombiner* (MIMV) is based on a variation of the general majority vote method. The fusion consists of a decision vector which results from the majority of assignments for each individual class.

combine(*decision_tensor*)

Combine decision outputs by MIMV across all classifiers per class (micro). Only crisp classification outputs are supported.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments obtained by MIMV. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.micro_majority_vote_combiner.**CRMicroMajorityVoteCombiner**

Bases: *pusion.core.micro_majority_vote_combiner.MicroMajorityVoteCombiner*

The *CRMicroMajorityVoteCombiner* is a modification of *MicroMajorityVoteCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To call *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

combine(*decision_outputs*)

Combine decision outputs by MIMV across all classifiers per class (micro). Only crisp classification outputs are supported.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which are obtained by MIMV. Axis 0 represents samples and axis 1 all the class labels which are provided by the coverage.

3.3.11 pusion.core.naive_bayes_combiner

class pusion.core.naive_bayes_combiner.**NaiveBayesCombiner**

Bases: *pusion.core.combiner.EvidenceBasedCombiner*, *pusion.core.combiner.TrainableCombiner*

The *NaiveBayesCombiner* (NB) is a fusion method based on the Bayes theorem which is applied according to Kuncheva[1] and Titterington et al.[2]. NB uses the confusion matrix as an evidence to calculate the a-priori probability and the bayesian belief value, which in turn the decision fusion bases on. NB requires outputs from uncorrelated classifiers in the ensemble.

[1] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

[2] DM Titterington, GD Murray, LS Murray, DJ Spiegelhalter, AM Skene, JDF Habbema, and GJ Gelpke. Comparison of discrimination techniques applied to a complex data set of head injured patients. *Journal of the Royal Statistical Society: Series A (General)*, 144(2):145–161, 1981.

set_evidence(*evidence*)
Set the evidence given by confusion matrices calculated according to Kuncheva[?] for each ensemble classifier.

Parameters **evidence** – *numpy.array* of shape *(n_classifiers, n_classes, n_classes)*. Confusion matrices for each of *n* classifiers.

train(*decision_tensor*, *true_assignments*)
Train the Naive Bayes combiner model by precalculating confusion matrices from given decision outputs and true class assignments. Continuous decision outputs are converted into crisp multiclass assignments using the MAX rule.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)
Combine decision outputs by using the Naive Bayes method. Continuous decision outputs are converted to crisp multiclass predictions using the MAX rule. Combining requires a trained *NaiveBayesCombiner* or evidence set with `set_evidence`.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which represents fused decisions obtained by the maximum class support. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.naive_bayes_combiner.**CRNaiveBayesCombiner**
Bases: *pusion.core.naive_bayes_combiner.NaiveBayesCombiner*

The *CRNaiveBayesCombiner* is a modification of *NaiveBayesCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as *0*, respectively. To call *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)
Train the Naive Bayes combiner model by precalculating confusion matrices from given decision outputs and true class assignments. Continuous decision outputs are converted into crisp multiclass assignments using the MAX rule.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)
Combine decision outputs by using the Naive Bayes method. Continuous decision outputs are converted to crisp multiclass predictions using the MAX rule. Combining requires a trained *NaiveBayesCombiner* or evidence set with `set_evidence`.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which represents fused decisions. Axis 0 represents samples and axis 1 the class labels which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.12 pusion.core.neural_network_combiner

class pusion.core.neural_network_combiner.**NeuralNetworkCombiner**

Bases: *pusion.core.combiner.TrainableCombiner*

The *NeuralNetworkCombiner* (NN) is a learning and classifier-based combiner that converts multiple decision outputs into new features, which in turn are used to train this combiner. The NN includes three hidden layers and a dynamic number of neurons per layer, which is given by (*n_classifiers* * *n_classes*).

train(*decision_tensor*, *true_assignments*)

Train the NN combiner by fitting the Neural Network model with given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by the trained Neural Network model. Both continuous and crisp classification outputs are supported. Combining requires a trained *NeuralNetworkCombiner*. This procedure transforms decision outputs into a new feature space.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by NN. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.neural_network_combiner.**CRNeuralNetworkCombiner**

Bases: *pusion.core.neural_network_combiner.NeuralNetworkCombiner*

The *CRNeuralNetworkCombiner* is a modification of *NeuralNetworkCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed, such that all missing classification assignments are considered as a constant, respectively. To use methods *train()* and *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

train(*decision_outputs*, *true_assignments*)

Train the NN combiner by fitting the Neural Network model with given decision outputs and true class assignments. Both continuous and crisp classification outputs are supported. This procedure transforms decision outputs into a new feature space.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_outputs*)

Combine decision outputs by the trained Neural Network model. Both continuous and crisp classification outputs are supported. Combining requires a trained *NeuralNetworkCombiner*. This procedure transforms decision outputs into a new feature space.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of either crisp or continuous class assignments which represents fused decisions obtained by NN. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.13 pusion.core.simple_average_combiner

class pusion.core.simple_average_combiner.**SimpleAverageCombiner**

Bases: *pusion.core.combiner.UtilityBasedCombiner*

The *SimpleAverageCombiner* (AVG) fuses decisions using the arithmetic mean rule. The mean is calculated between decision vectors obtained by multiple ensemble classifiers for a sample. The AVG combiner is unaware of the input problem (multiclass/multilabel) or the assignment type (crisp/continuous).

combine(*decision_tensor*)

Combine decision outputs by averaging the class support of each classifier in the given ensemble.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp assignments which represents fused decisions obtained by the AVG method. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.simple_average_combiner.**CRSimpleAverageCombiner**

Bases: *pusion.core.simple_average_combiner.SimpleAverageCombiner*

The *CRSimpleAverageCombiner* is a modification of *SimpleAverageCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed to a unified tensor representation supporting undefined class assignments. The mean is calculated only for assignments which are defined. To call *combine()* a coverage needs to be set first by the inherited `set_coverage()` method.

combine(*decision_outputs*)

Combine decision outputs by averaging the defined class support of each classifier in the given ensemble. Undefined class supports are excluded from averaging.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains either crisp or continuous decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp assignments which represents fused decisions obtained by the AVG method. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

3.3.14 pusion.core.weighted_voting_combiner

class pusion.core.weighted_voting_combiner.**WeightedVotingCombiner**

Bases: *pusion.core.combiner.EvidenceBasedCombiner*, *pusion.core.combiner.TrainableCombiner*

The *WeightedVotingCombiner* (WV) is a weighted voting schema adopted from Kuncheva (eq. 4.43)[1]. Classifiers with better performance (i.e. accuracy) are given more weight contributing to final decisions. Nevertheless, if classifiers of high performance disagree on a sample, low performance classifiers may contribute to the final decision.

set_evidence(*evidence*)

Set the evidence given by confusion matrices calculated according to Kuncheva[?] for each ensemble classifier.

Parameters **evidence** – *numpy.array* of shape *(n_classifiers, n_classes, n_classes)*. Confusion matrices for each of *n* classifiers.

train(*decision_tensor*, *true_assignments*)

Train the Weighted Voting combiner model by precalculating confusion matrices from given decision outputs and true class assignments. Continuous decision outputs are converted into crisp multiclass assignments using the MAX rule.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of either crisp or continuous class assignments which are considered true for each sample during the training procedure.

combine(*decision_tensor*)

Combine decision outputs by the weighted voting schema. Classifiers with better performance (i.e. accuracy) are given more authority over final decisions. Combining requires a trained *WeightedVotingCombiner* or evidence set with `set_evidence`.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which represents fused decisions obtained by the maximum weighted class support. Axis 0 represents samples and axis 1 the class assignments which are aligned with axis 2 in `decision_tensor` input tensor.

class pusion.core.weighted_voting_combiner.**CRWeightedVotingCombiner**

Bases: *pusion.core.weighted_voting_combiner.WeightedVotingCombiner*

The *CRWeightedVotingCombiner* is a modification of *WeightedVotingCombiner* that also supports complementary-redundant decision outputs. Therefore the input is transformed to a unified tensor representation supporting undefined class assignments. The mean is calculated only for assignments which are defined. To call methods *train()* and *combine()*, a coverage needs to be set first by the inherited `set_coverage()` method.

[1] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

train(*decision_outputs*, *true_assignments*)
Train the Weighted Voting combiner model by precalculating confusion matrices from given decision outputs and true class assignments. Continuous decision outputs are converted into crisp multiclass assignments using the MAX rule.

Parameters

- **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp decision outputs per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which is considered true for each sample during the training procedure.

combine(*decision_outputs*)
Combine decision outputs by the weighted voting schema. Classifiers with better performance (i.e. accuracy) are given more authority over final decisions. Combining requires a trained *WeightedVotingCombiner* or evidence set with `set_evidence`.

Parameters **decision_outputs** – *list* of *numpy.array* matrices, each of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific and described by the coverage. Each matrix corresponds to one of *n_classifiers* classifiers and contains crisp decision outputs per sample.

Returns A matrix (*numpy.array*) of crisp class assignments which are obtained by the best representative class for a certain classifier's behaviour per sample. Axis 0 represents samples and axis 1 all the class labels which are provided by the coverage.

3.4 pusion.evaluation package

3.4.1 pusion.evaluation.evaluation module

class pusion.evaluation.evaluation.**Evaluation**(**argv*)
Bases: `object`

Evaluation provides methods for evaluating decision outputs (i.e. combiners and classifiers) with different problems and coverage types.

Parameters **argv** – Performance metric functions.

evaluate(*true_assignments*, *decision_tensor*)
Evaluate the decision outputs with already set classification performance metrics.

> **Warning:** This evaluation is only applicable on redundant multiclass or multilabel decision outputs.

Parameters

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for the evaluation.
- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_instances, n_metrics)*. Performance matrix containing performance values for each set instance row-wise and each set performance metric column-wise.

evaluate_cr_decision_outputs(*true_assignments*, *decision_outputs*, *coverage=None*)

Evaluate complementary-redundant decision outputs with already set classification performance metrics. The outputs of each classifier for each class is considered as a binary output and thus, the performance is calculated class-wise and averaged across all classes, which are covered by individual classifiers.

Note: This evaluation is applicable on complementary-redundant ensemble classifier outputs.

Parameters

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for the evaluation.
- **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list. If none set, the coverage for fully redundant classification is chosen by default.

Returns *numpy.array* of shape *(n_instances, n_metrics)*. Performance matrix containing performance values for each set instance row-wise and each set performance metric column-wise.

evaluate_cr_multi_combiner_decision_outputs(*true_assignments*, *decision_tensor*)

Evaluate decision outputs of multiple CR combiners with already set classification performance metrics. The evaluation is performed by *evaluate_cr_decision_outputs()* for each combiner.

Parameters

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for the evaluation.
- **decision_tensor** – *numpy.array* of shape *(n_combiners, n_samples, n_classes)*. Tensor of crisp decision outputs by different combiners per sample.

Returns *numpy.array* of shape *(n_instances, n_metrics)*. Performance matrix containing performance values for each set instance row-wise and each set performance metric column-wise.

class_wise_mean_score(*true_assignments*, *decision_outputs*, *coverage*, *metric*)

Calculate the class-wise mean score with the given metric for the given classification outputs.

Parameters

- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for the evaluation.
- **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list. If none set, the coverage for fully redundant classification is chosen by default.
- **metric** – The score metric.

Returns *numpy.array* of shape *(n_classes,)*. The mean score per class across all classifiers.

get_report()

Returns A summary *Report* of performed evaluations including all involved instances and performance metrics.

`get_runtime_report`()

Returns A summary *Report* of train and combine runtimes for all involved instances.

`get_instances`()

Returns A *list* of instances (i.e. combiner or classifiers) been evaluated.

`get_metrics`()

Returns A *list* of performance metrics been used for evaluation.

`get_performance_matrix`()

Returns *numpy.array* of shape *(n_instances, n_metrics)*. Performance matrix containing performance values for each set instance row-wise and each set performance metric column-wise.

`get_runtime_matrix`()

Returns *numpy.array* of shape *(n_instances, 2)*. Runtime matrix containing runtimes for each set instance row-wise. The column at index *0* describes train times and the column at index *1* describes combine times.

`get_top_n_instances`(*n=None*, *metric=None*)
Retrieve top *n* best instances according to the given *metric* in a sorted order.

Parameters

- **n** – *integer*. Number of instances to be retrieved. If unset, all instances are retrieved.
- **metric** – The metric all instances are sorted by. If unset, the first metric is used.

Returns Evaluated top *n* instances.

`get_top_instances`(*metric=None*)
Retrieve best performing instances according to the given *metric*. Multiple instances may be returned having the identical best performance score.

Parameters **metric** – The metric all instances were evaluated with. If unset, the first metric is used.

Returns Evaluated top instances according to their performance.

`get_instance_performance_tuples`(*metric=None*)
Retrieve (instance, performance) tuples created for to the given *metric*.

Parameters **metric** – The metric all instances are evaluated by. If unset, the first set metric is used.

Returns *list* of (instance, performance) tuples.

`set_metrics`(**argv*)

Parameters **argv** – Performance metric functions.

`set_instances`(*instances*)

Parameters **instances** – An instance or a *list* of instances to be evaluated, e.g. classifiers or combiners.

set_runtimes(*runtimes*)

Parameters **runtimes** – A *tuple* of two lists of tuples describing the train and combine runtimes respectively. Each runtime list is aligned with the list of set instances.

3.4.2 pusion.evaluation.evaluation_metrics module

pusion.evaluation.evaluation_metrics.**multi_label_brier_score_micro**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Calculate the brier score for multi-label problems according to Brier 1950 :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The micro brier score.

pusion.evaluation.evaluation_metrics.**multi_label_brier_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Calculate the brier score for multiclass problems according to Brier 1950 :param y_true: *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The brier score.

pusion.evaluation.evaluation_metrics.**multiclass_brier_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Calculate the brier score for multi-label problems according to Brier 1950 :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The brier score.

pusion.evaluation.evaluation_metrics.**far**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*, *pos_normal_class: int = 0*) → float

Calculate the false alarm rate for multiclass and multi-label problems. FAR = (number of normal class samples incorrectly classified)/(number of all normal class samples) * 100 :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :param pos_normal_class: the position of the 'normal class' in :param y_true and :param y_pred. Default is *0* :return: The false alarm rate.

pusion.evaluation.evaluation_metrics.**multiclass_fdr**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

fault detection rate = (# correctly classified faulty samples) / (# all faulty samples) * 100 :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The fault detection rate.

pusion.evaluation.evaluation_metrics.**multilabel_subset_fdr**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

fault detection rate = (# correctly classified faulty samples) / (# all faulty samples) * 100 In multilabel classification, the function considers the faulty subset, i. e., if the entire set of predicted faulty labels for a sample strictly match with the true set of faulty labels. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The fault detection rate.

pusion.evaluation.evaluation_metrics.**multilabel_minor_fdr**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

fault detection rate = (# correctly classified faulty samples) / (# all faulty samples) * 100 In multilabel classification, the function considers the faulty subset, i. e., if the entire set of predicted faulty labels for a sample strictly match with the true set of faulty labels. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True

labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The fault detection rate.

pusion.evaluation.evaluation_metrics.**multiclass_weighted_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Calculate the precision for a multiclass problem with a *weighted* average. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples,)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multi_label_weighted_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Calculate the precision for a multi-label problem with a *weighted* average. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multiclass_class_wise_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → numpy.ndarray

Calculate the precision for a multiclass problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multi_label_class_wise_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → numpy.ndarray

Calculate the precision for a multi-label problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multiclass_recall**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Calculate the recall for a multiclass problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples,)*. Predicted labels or class assignments. :return: The recall score.

pusion.evaluation.evaluation_metrics.**multi_label_recall**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Calculate the recall for a multi-label problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The recall score.

pusion.evaluation.evaluation_metrics.**multiclass_class_wise_recall**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → numpy.ndarray

Calculate the recall for a multiclass problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: Sequence of recall scores (for each class).

pusion.evaluation.evaluation_metrics.**multi_label_class_wise_recall**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → numpy.ndarray

Calculate the recall for a multi-label problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: Sequence of recall scores (for each class).

pusion.evaluation.evaluation_metrics.**multiclass_weighted_scikit_auc_roc_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the scikit auc roc score for a multi-label problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The auc roc score.

pusion.evaluation.evaluation_metrics.**multi_label_weighted_pytorch_auc_roc_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the pytorch auc roc score for a multi-label problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The auc roc score.

pusion.evaluation.evaluation_metrics.**multi_label_pytorch_auc_roc_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Compute the pytorch auc roc score for a multi-label problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The auc roc score.

pusion.evaluation.evaluation_metrics.**multiclass_class_wise_avg_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Compute the class wise precision for a multiclass problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multiclass_weighted_avg_precision**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the precision for a multiclass problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multiclass_auc_precision_recall_curve**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Compute the class wise auc precision recall curve for a multiclass problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The aggregated auc precision recall curve class wise.

pusion.evaluation.evaluation_metrics.**multiclass_weighted_pytorch_auc_roc**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the pytorch auc roc for a multiclass problem with average *weighted*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The auc roc score.

pusion.evaluation.evaluation_metrics.**multiclass_pytorch_auc_roc**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*)

Compute the pytorch auc roc for a multiclass problem with average *None*. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The auc roc score.

pusion.evaluation.evaluation_metrics.**multi_label_ranking_avg_precision_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the label ranking based average precision score for a multi-label problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score.

pusion.evaluation.evaluation_metrics.**multi_label_ranking_loss**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the label ranking loss for a multi-label problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The precision score. :return: The loss.

pusion.evaluation.evaluation_metrics.**multi_label_normalized_discounted_cumulative_gain**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the normalized discounted cumulative gain for a multi-label problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The gain.

pusion.evaluation.evaluation_metrics.**multiclass_top_1_accuracy**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the top-1 accuracy for a multiclass problem. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The accuracy score.

pusion.evaluation.evaluation_metrics.**multiclass_top_3_accuracy**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the top-3 accuracy for a multiclass problem. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The accuracy score.

pusion.evaluation.evaluation_metrics.**multiclass_top_5_accuracy**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Compute the top-5 accuracy for a multiclass problem. :param y_true: *numpy.array* of shape *(n_samples,)*. True labels or class assignments. :param y_pred: *numpy.array* of shape (n_samples, n_classes)`. Predicted labels or class assignments. :return: The accuracy score.

pusion.evaluation.evaluation_metrics.**multiclass_log_loss**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

The logarithmic loss for a multiclass problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The loss.

pusion.evaluation.evaluation_metrics.**multi_label_log_loss**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

The logarithmic loss for a multi-label problem. :param y_true: *numpy.array* of shape *(n_samples, n_classes)*.

True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples, n_classes)*. Predicted labels or class assignments. :return: The loss.

pusion.evaluation.evaluation_metrics.**micro_precision**(*y_true*, *y_pred*)
Calculate the micro precision, i.e. TP / (TP + FP).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The micro precision.

pusion.evaluation.evaluation_metrics.**micro_recall**(*y_true*, *y_pred*)
Calculate the micro recall, i.e. TP / (TP + FN).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The micro recall.

pusion.evaluation.evaluation_metrics.**micro_f1**(*y_true*, *y_pred*)
Calculate the micro F1-score, i.e. 2 * (Precision * Recall) / (Precision + Recall).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The micro F1-score.

pusion.evaluation.evaluation_metrics.**micro_f2**(*y_true*, *y_pred*)
Calculate the micro F2-score (beta=2).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The micro F2-score.

pusion.evaluation.evaluation_metrics.**micro_jaccard**(*y_true*, *y_pred*)
Calculate the micro Jaccard-score, i.e. TP / (TP + FP + FN).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The micro Jaccard-score.

pusion.evaluation.evaluation_metrics.**macro_precision**(*y_true*, *y_pred*)
Calculate the macro precision, i.e. TP / (TP + FP).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro precision.

pusion.evaluation.evaluation_metrics.**macro_recall**(*y_true*, *y_pred*)
Calculate the macro recall, i.e. TP / (TP + FN).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro recall.

pusion.evaluation.evaluation_metrics.**macro_f1**(*y_true*, *y_pred*)
Calculate the macro F1-score, i.e. 2 * (Precision * Recall) / (Precision + Recall).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro F1-score.

pusion.evaluation.evaluation_metrics.**weighted_f1**(*y_true*, *y_pred*)
Calculate the macro F1-score, i.e. 2 * (Precision * Recall) / (Precision + Recall), weighted by the class support.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The weighted macro F1-score.

pusion.evaluation.evaluation_metrics.**macro_f2**(*y_true*, *y_pred*)
Calculate the macro F2-score (beta=2).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro F2-score.

pusion.evaluation.evaluation_metrics.**macro_jaccard**(*y_true*, *y_pred*)
Calculate the macro Jaccard-score, i.e. TP / (TP + FP + FN).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro Jaccard-score.

pusion.evaluation.evaluation_metrics.**weighted_jaccard**(*y_true*, *y_pred*)
Calculate the Jaccard-score for each label, and find their average, weighted by support, i. e., the number of true instances of each label instance.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns The macro Jaccard-score.

pusion.evaluation.evaluation_metrics.**accuracy**(*y_true*, *y_pred*)
Calculate the accuracy, i.e. (TP + TN) / (TP + FP + FN + TN).

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns Accuracy.

pusion.evaluation.evaluation_metrics.**error_rate**(*y_true*, *y_pred*)
Calculate the error rate, i. e. error_rate = 1-accuracy :param y_true: *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments. :param y_pred: *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments. :return: Error Rate of typ *float*

pusion.evaluation.evaluation_metrics.**balanced_multiclass_accuracy**(*y_true*, *y_pred*)
Calculate the balanced accuracy, i.e. (Precision + Recall) / 2.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns Accuracy.

pusion.evaluation.evaluation_metrics.**mean_multilabel_confusion_matrix**(*y_true*, *y_pred*)
Calculate the normalized mean confusion matrix across all classes.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.

- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns *numpy.array* of shape *(n_classes, n_classes)*. Normalized mean confusion matrix.

pusion.evaluation.evaluation_metrics.**mean_confidence**(*y_true*, *y_pred*)
Calculate the mean confidence for continuous multiclass and multilabel classification outputs.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. True class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples, n_classes)*. Predicted class assignments.

Returns Mean confidence.

pusion.evaluation.evaluation_metrics.**hamming**(*y_true*, *y_pred*)
Calculate the average Hamming Loss.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns Average Hamming Loss.

pusion.evaluation.evaluation_metrics.**log**(*y_true*, *y_pred*)
Calculate the Logistic Loss.

Parameters

- **y_true** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. True labels or class assignments.
- **y_pred** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Predicted labels or class assignments.

Returns Logistic Loss.

pusion.evaluation.evaluation_metrics.**cohens_kappa**(*y1*, *y2*, *labels*)
Calculate the Cohen's Kappa annotator agreement score according to[1].

Parameters

- **y1** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Labels or class assignments.
- **y2** – *numpy.array* of shape *(n_samples,)* or *(n_samples, n_classes)*. Labels or class assignments.
- **labels** – *list* of all possible labels.

Returns Cohen's Kappa score.

pusion.evaluation.evaluation_metrics.**pairwise_cohens_kappa**(*decision_tensor*)
Calculate the average of pairwise Cohen's Kappa scores over all multiclass decision outputs. E.g., for 3 classifiers *(0,1,2)*, the agreement score is calculated for classifier tuples *(0,1)*, *(0,2)* and *(1,2)*. These scores are then averaged over all 3 classifiers.

[1] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.

Returns Pairwise (averages) Cohen's Kappa score.

pusion.evaluation.evaluation_metrics.**correlation**(*y1*, *y2*, *y_true*)

Calculate the correlation score for decision outputs of two classifiers according to Kuncheva[2].

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Correlation score.

pusion.evaluation.evaluation_metrics.**q_statistic**(*y1*, *y2*, *y_true*)

Calculate the Q statistic score for decision outputs of two classifiers according to Yule[3].

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Correlation score.

pusion.evaluation.evaluation_metrics.**kappa_statistic**(*y1*, *y2*, *y_true*)

Calculate the kappa score for decision outputs of two classifiers according to Kuncheva[?].

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Kappa score.

pusion.evaluation.evaluation_metrics.**disagreement**(*y1*, *y2*, *y_true*)

Calculate the disagreement for decision outputs of two classifiers, i.e. the percentage of samples which are correctly classified by exactly one of the classifiers.

[2] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

[3] G Udny Yule. On the association of attributes in statistics: with illustrations from the material of the childhood society, &c. *Philosophical Transactions of the Royal Society of London Series A*, 194:257–319, 1900.

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Disagreement score.

pusion.evaluation.evaluation_metrics.**double_fault**(*y1*, *y2*, *y_true*)

Calculate the double fault for decision outputs of two classifiers, i.e. the percentage of samples which are misclassified by both classifiers.

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Double fault score.

pusion.evaluation.evaluation_metrics.**abs_correlation**(*y1*, *y2*, *y_true*)

Calculate the absolute correlation score for decision outputs of two classifiers.

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Correlation score.

pusion.evaluation.evaluation_metrics.**abs_q_statistic**(*y1*, *y2*, *y_true*)

Calculate the absolute Q statistic score for decision outputs of two classifiers.

Parameters

- **y1** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the first classifier.
- **y2** – *numpy.array* of shape *(n_samples, n_classes)*. Crisp multiclass decision outputs by the second classifier.
- **y_true** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Correlation score.

pusion.evaluation.evaluation_metrics.**pairwise_correlation**(*decision_tensor*, *true_assignments*, ***kwargs*)

Calculate the average of the pairwise absolute correlation scores over all decision outputs.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Pairwise correlation score.

pusion.evaluation.evaluation_metrics.**pairwise_q_statistic**(*decision_tensor*, *true_assignments*)

Calculate the average of the pairwise absolute Q-statistic scores over all decision outputs.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Pairwise correlation score.

pusion.evaluation.evaluation_metrics.**pairwise_kappa_statistic**(*decision_tensor*, *true_assignments*, ***kwargs*)

Calculate the average of pairwise Kappa scores over all decision outputs. Multilabel class assignments are transformed to equivalent multiclass class assignments.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Pairwise kappa score.

pusion.evaluation.evaluation_metrics.**pairwise_disagreement**(*decision_tensor*, *true_assignments*)

Calculate the average of pairwise disagreement scores over all decision outputs. Multilabel class assignments are transformed to equivalent multiclass class assignments.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Pairwise disagreement score.

pusion.evaluation.evaluation_metrics.**pairwise_double_fault**(*decision_tensor*, *true_assignments*, ***kwargs*)

Calculate the average of pairwise double fault scores over all decision outputs. Multilabel class assignments are transformed to equivalent multiclass class assignments.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered as true.

Returns Pairwise double fault score.

pusion.evaluation.evaluation_metrics.**pairwise_euclidean_distance**(*decision_tensor*)

Calculate the average of pairwise euclidean distance between decision matrices for the given classifiers.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.

Returns Pairwise euclidean distance.

3.5 pusion.model package

3.5.1 pusion.model.configuration module

class pusion.model.configuration.**Configuration**(*method*, *problem='GENERIC'*, *assignment_type='GENERIC'*, *coverage_type='GENERIC'*)

Bases: object

The *Configuration* forms the main parameter of the decision fusion framework. Based on this, the framework is able to check the compatibility of a respective decision fusion method to the given decision outputs. A configuration may be defined by the user or auto-detected by the framework.

Parameters

- **method** – An explicit method provided by the framework which should be applied on the input data set. See *pusion.Method* for possible options.
- **problem** – Input problem type. See *pusion.util.constants.Problem* for possible options.
- **assignment_type** – The class assignment describes memberships to each individual class for a sample. A class assignment type is either crisp or continuous. Crisp assignments are equivalent to labels and continuous assignments represent probabilities for each class being true. See *pusion.util.constants.AssignmentType* for possible options.
- **coverage_type** – The classification coverage states for each input classifier, which classes it is able to decide. A classifier ensemble may yield a redundant, complementary or complementary-redundant coverage. See *pusion.util.constants.CoverageType* for possible options.

get_tuple()

Returns A *tuple* of method, problem, assignment type and coverage type.

get_pac()

Returns A *tuple* of problem, assignment type and coverage type. This tuple is also referred to as PAC.

3.5.2 pusion.model.report module

class pusion.model.report.**Report**(*performance_matrix*, *instances*, *metrics*)
Bases: object

Report is a string representation of the performance matrix retrieved by Evaluation methods.

Parameters

- **performance_matrix** – *numpy.array* of shape *(n_instances, n_metrics)*. Performance matrix containing performance values for each set instance row-wise and each set performance metric column-wise.
- **instances** – *list* of instances been evaluated which is aligned with the performance_matrix on axis *0*.
- **metrics** – *list* of metric functions which is aligned with the performance_matrix on axis *1*.

3.6 pusion.util package

3.6.1 pusion.util.generator module

pusion.util.generator.**generate_multiclass_ensemble_classification_outputs**(*classifiers*, *n_classes*, *n_samples*, *continuous_out=False*, *parallelize=True*)

Generate random multiclass, crisp and redundant classification outputs (assignments) for the given ensemble of classifiers.

Parameters

- **classifiers** – Classifiers used to generate classification outputs. These need to implement *fit* and *predict* methods according to classifiers provided by *sklearn*.
- **n_classes** – *integer*. Number of classes, predictions are made for.
- **n_samples** – *integer*. Number of samples.
- **parallelize** – If *True*, all classifiers are trained in parallel. Otherwise they are trained in sequence.
- **continuous_out** – If *True*, class assignments in *y_ensemble_valid* and *y_ensemble_test* are given as probabilities. Default value is *False*.

Returns *tuple* of: - *y_ensemble_valid*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a validation dataset. - *y_valid*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the validation. - *y_ensemble_test*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a test dataset. - *y_test*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the test.

pusion.util.generator.**generate_multiclass_cr_ensemble_classification_outputs**(*classifiers*, *n_classes*, *n_samples*, *coverage=None*, *continuous_out=False*, *parallelize=True*)

Generate random multiclass, crisp and complementary-redundant classification outputs (assignments) for the given ensemble of classifiers.

Parameters

- **classifiers** – Classifiers used to generate classification outputs. These need to implement *fit* and *predict* methods according to classifiers provided by *sklearn*.
- **n_classes** – *integer*. Number of classes, predictions are made for.
- **n_samples** – *integer*. Number of samples.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list. If unset, redundant classification outputs are retrieved.
- **continuous_out** – If *True*, class assignments in *y_ensemble_valid* and *y_ensemble_test* are given as probabilities. Default value is *False*.
- **parallelize** – If *True*, all classifiers are trained in parallel. Otherwise they are trained in sequence.

Returns *tuple* of: - *y_ensemble_valid*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a validation dataset. - *y_valid*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the validation. - *y_ensemble_test*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a test dataset. - *y_test*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the test.

pusion.util.generator.**generate_multilabel_ensemble_classification_outputs**(*classifiers*, *n_classes*, *n_samples*, *continuous_out=False*, *parallelize=True*)

Generate random multilabel crisp classification outputs (assignments) for the given ensemble of classifiers with the normal class included at index *0*.

Parameters

- **classifiers** – Classifiers used to generate classification outputs. These need to implement *fit* and *predict* methods according to classifiers provided by *sklearn*.
- **n_classes** – *integer*. Number of classes, predictions are made for with the normal class included.
- **n_samples** – *integer*. Number of samples.
- **continuous_out** – If *True*, class assignments in *y_ensemble_valid* and *y_ensemble_test* are given as probabilities. Default value is *False*.
- **parallelize** – If *True*, all classifiers are trained in parallel. Otherwise they are trained in sequence.

Returns *tuple* of: - *y_ensemble_valid*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a validation dataset. - *y_valid*: *numpy.array* of shape *(n_samples,*

n_classes). True class assignments for the validation. - *y_ensemble_test*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a test dataset. - *y_test*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the test.

pusion.util.generator.**generate_multilabel_cr_ensemble_classification_outputs**(*classifiers*, *n_classes*, *n_samples*, *coverage=None*, *continuous_out=False*, *parallelize=True*)

Generate random multilabel, crisp and complementary-redundant classification outputs (assignments) for the given ensemble of classifiers with the normal class included at index *0*.

Parameters

- **classifiers** – Classifiers used to generate classification outputs. These need to implement *fit* and *predict* methods according to classifiers provided by *sklearn*.
- **n_classes** – *integer*. Number of classes, predictions are made for with the normal class included.
- **n_samples** – *integer*. Number of samples.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list. If unset, redundant classification outputs are retrieved.
- **continuous_out** – If *True*, class assignments in *y_ensemble_valid* and *y_ensemble_test* are given as probabilities. Default value is *False*.
- **parallelize** – If *True*, all classifiers are trained in parallel. Otherwise they are trained in sequence.

Returns *tuple* of: - *y_ensemble_valid*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a validation dataset. - *y_valid*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the validation. - *y_ensemble_test*: *numpy.array* of shape *(n_samples, n_classes)*. Ensemble decision output matrix for as a test dataset. - *y_test*: *numpy.array* of shape *(n_samples, n_classes)*. True class assignments for the test.

pusion.util.generator.**generate_multiclass_confusion_matrices**(*decision_tensor*, *true_assignments*)

Generate multiclass confusion matrices out of the given decision tensor and true assignments. Continuous outputs are converted to multiclass assignments using the MAX rule.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for calculating confusion matrices.

Returns *numpy.array* of shape *(n_classifiers, n_samples, n_samples)*. Confusion matrices per classifier.

pusion.util.generator.**generate_multilabel_cr_confusion_matrices**(*decision_outputs*, *true_assignments*, *coverage*)

Generate multilabel confusion matrices for complementary-redundant multilabel classification outputs.

Parameters

- **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of crisp class assignments which are considered true for calculating confusion matrices.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

Returns List of multilabel confusion matrices.

pusion.util.generator.**generate_classification_coverage**(*n_classifiers*, *n_classes*, *overlap*, *normal_class=True*)

Generate random complementary redundant class indices for each classifier *0..(n_classifiers-1)*. The coverage is drawn from normal distribution for all classifiers. However, it is guaranteed that each classifier covers at least one class regardless of the distribution.

Parameters

- **n_classifiers** – Number of classifiers representing the classifier *0..(n_classifiers-1)*.
- **n_classes** – Number of classes representing the class label *0..(n_classes-1)*.
- **overlap** – Indicator between *0* and *1* for overall classifier overlapping in terms of classes. If *0*, only complementary class indices are obtained. If *1*, the overlapping is fully redundant.
- **normal_class** – If *True*, a class for the normal state is included for all classifiers as class index *0*.

Returns *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

pusion.util.generator.**shrink_to_coverage**(*decision_tensor*, *coverage*)

Shrink the given decision tensor to decision outputs according to the given coverage. Assumption: the normal class is covered by each classifier at index *0*.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multilabel decision outputs by different classifiers per sample.
- **coverage** – *list* of *list* elements. Each inner list contains classes as integers covered by a classifier, which is identified by the positional index of the respective list.

Returns *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

pusion.util.generator.**split_into_train_and_validation_data**(*decision_tensor*, *true_assignments*, *validation_size=0.5*)

Split the decision outputs (tensor) from multiple classifiers as well as the true assignments randomly into train and validation datasets.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.
- **true_assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of true class assignments.
- **validation_size** – Proportion between *0* and *1* for the size of the validation data set.

Returns *tuple* of (1) *numpy.array* of shape *(n_classifiers, n_samples', n_classes)*, (2) *numpy.array* of shape *(n_classifiers, n_samples')*, (3) *numpy.array* of shape *(n_classifiers, n_samples", n_classes)*, (4) *numpy.array* of shape *(n_classifiers, n_samples")*, with *n_samples'* as the number of training samples and *n_samples"* as the number of validation samples.

3.6.2 pusion.util.transformer module

pusion.util.transformer.**confusion_matrices_to_accuracy_vector**(*confusion_matrix_tensor*)
Convert confusion matrices of respective classification to an accuracy vector.

Parameters **confusion_matrix_tensor** – *numpy.array* of shape *(n_classifiers, n_classes, n_classes)* Confusion matrices.

Returns One-dimensional *numpy.array* of shape of length *n_classifiers* containing the accuracy for each confusion matrix.

pusion.util.transformer.**confusion_matrix_to_accuracy**(*cm*)
Calculate the accuracy out of the given confusion matrix.

Parameters **cm** – *numpy.array* of shape *(n_classes, n_classes)*. Confusion matrix.

Returns The accuracy.

pusion.util.transformer.**multilabel_cr_confusion_matrices_to_avg_accuracy**(*label_cms*)
Calculate the average accuracy for the given confusion matrices generated from complementary redundant multilabel output.

Parameters **label_cms** – *list* of confusion matrices given as 2-dimensional *numpy.array* respectively.

Returns The average accuracy.

pusion.util.transformer.**decision_tensor_to_decision_profiles**(*decision_tensor*)
Transform the given decision tensor to decision profiles for each respective sample.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of either crisp or continuous decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_samples, n_classifiers, n_classes)*. Decision profiles.

pusion.util.transformer.**multilabel_predictions_to_decisions**(*predictions*, *threshold=0.5*)
Transform a continuously valued tensor of multilabel decisions to crisp decision outputs.

Parameters

- **predictions** – *numpy.array* of any shape. Continuous predictions.
- **threshold** – *float*. A threshold value, based on which the crisp output is constructed.

Returns *numpy.array* of the same shape as `predictions`. Crisp decision outputs.

pusion.util.transformer.**multiclass_predictions_to_decisions**(*predictions*)
Transform a continuously valued matrix of multiclass decisions to crisp decision outputs.

Parameters **predictions** – *numpy.array* of shape *(n_samples, n_classes)*. Continuous predictions.

Returns *numpy.array* of the same shape as `predictions`. Crisp decision outputs.

pusion.util.transformer.**multilabel_prediction_tensor_to_decision_tensor**(*predictions*)
Transform a continuously valued tensor of multilabel decisions to crisp decision outputs.

Parameters **predictions** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Continuous predictions.

Returns *numpy.array* of the same shape as `predictions`. Crisp decision outputs.

pusion.util.transformer.**multiclass_prediction_tensor_to_decision_tensor**(*predictions*)
Transform a continuously valued tensor of multiclass decisions to crisp decision outputs.

Parameters **predictions** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Continuous predictions.

Returns *numpy.array* of the same shape as `predictions`. Crisp decision outputs.

pusion.util.transformer.**decision_tensor_to_configs**(*decision_outputs*)
Transform decision outputs to decision configs. A decision config shows concatenated classification outputs of each classifier per sample.

Parameters **decision_outputs** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)* or a *list* of *numpy.array* elements of shape *(n_samples, n_classes')*, where *n_classes'* is classifier-specific due to the coverage.

Returns *numpy.array* of shape *(n_samples, n_classes*)*, *n_classes** is the sum of all classes covered by all classifiers.

pusion.util.transformer.**multiclass_assignments_to_labels**(*assignments*)
Transform multiclass assignments to labels. A matrix of shape *(n_samples, n_classes)* is converted to a vector of shape *(n_samples,)*, with element-wise labels represented in integers from *0* to *n_classes - 1*.

Parameters **assignments** – *numpy.array* of shape *(n_samples, n_classes)*. Multiclass assignments.

Returns *numpy.array* of shape *(n_samples,)* with an integer label per element.

pusion.util.transformer.**transform_label_tensor_to_class_assignment_tensor**(*label_tensor*, *n_classes*)
Transform a label tensor of shape *(n_classifiers, n_samples)* to the tensor of class assignments of shape *(n_classifiers, n_samples, n_classes)*. A label is an integer between *0* and *n_classes - 1*.

Parameters

- **label_tensor** – *numpy.array* of shape *(n_classifiers, n_samples)*. Label tensor.
- **n_classes** – Number of classes to be considered.

Returns *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Class assignment tensor (decision tensor).

pusion.util.transformer.**transform_label_vector_to_class_assignment_matrix**(*label_vector*, *n_classes=None*)
Transform labels to multiclass assignments. A vector of shape *(n_samples,)*, with element-wise labels is converted to the assignment matrix of shape *(n_samples, n_classes)*.

Parameters

- **label_vector** – *numpy.array* of shape *(n_samples,)* with an integer label per element.
- **n_classes** – Number of classes to be considered.

Returns *numpy.array* of shape *(n_samples, n_classes)*. Multiclass assignments.

pusion.util.transformer.**class_assignment_tensor_to_label_tensor**(*class_assignment_tensor*)
Transform multiclass class assignments into a label tensor. A tensor of shape *(n_classifiers, n_samples, n_classes)* is converted into a tensor of shape *(n_classifiers, n_samples)* holding the class identifier (label) in each element.

Parameters **class_assignment_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Class assignment tensor (decision tensor).

Returns *numpy.array* of shape *(n_classifiers, n_samples)*. Label tensor.

pusion.util.transformer.**class_assignment_matrix_to_label_vector**(*class_assignment_matrix*)
Transform multiclass class assignments into a label vector. A matrix of shape *(n_samples, n_classes)* is converted into a vector of shape *(n_samples,)* holding the class identifier (label) in each element.

Parameters **class_assignment_matrix** – *numpy.array* of shape *(n_samples, n_classes)*. Class assignment matrix (decision matrix).

Returns *numpy.array* of shape *(n_samples,)*. Label vector.

pusion.util.transformer.**multilabel_to_multiclass_assignments**(*decision_tensor*)
Transform the multilabel decision tensor to the equivalent multiclass decision tensor using the power set method. The multilabel class assignments are considered as a binary number which represents a new class in the multiclass decision space. E.g. the assignment to the classes *0* and *2* (*[1,0,1]*) is converted to the class *5*, which is one of the *2^3* classes in the multiclass decision space. This method is inverse to the `multiclass_to_multilabel_assignments` method.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multilabel decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_classifiers, n_samples, 2^n_classes)*. Tensor of crisp multiclass decision outputs by different classifiers per sample.

pusion.util.transformer.**multiclass_to_multilabel_assignments**(*decision_tensor*)
Transform the multiclass decision tensor to the equivalent multilabel decision tensor using the inverse power set method. The multiclass assignment is considered as a decimal which is converted to a binary number, which in turn represents the multilabel class assignment. E.g. the class assignment to the class *3 ([0,0,0,1])* is converted to the multilabel class assignment *[1,1]* (classes *0* and *1* in the multilabel decision space). This method is inverse to the `multilabel_to_multiclass_assignments` method.

Parameters **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of crisp multilabel decision outputs by different classifiers per sample.

Returns *numpy.array* of shape *(n_classifiers, n_samples, log_2(n_classes))*. Tensor of crisp multiclass decision outputs by different classifiers per sample.

pusion.util.transformer.**tensorize**(*decision_outputs*)
Convert *list* decision outputs to *numpy.array* decision tensor, if possible.

pusion.util.transformer.**intercept_normal_class**(*y*, *override=False*)
Intercept the normal class for the given decision matrix, i.e. a normal class is assigned to each zero vector class assignment. E.g. the assignment *[0,0,0,0]* is transformed to *[1,0,0,0]*, under the assumption that *0* is a normal class.

Parameters

- **y** – *numpy.array* of shape *(n_samples, n_classes)*. Matrix of decision outputs.
- **override** – If *true*, the class *0* is assumed as a normal class. Otherwise a new class is prepended to existing classes.

Returns *numpy.array* of shape *(n_samples, n_classes)* for *override=False*. *numpy.array* of shape *(n_samples, n_classes + 1)* for *override=True*. Matrix of decision outputs with intercepted normal class.

pusion.util.transformer.**intercept_normal_class_in_tensor**(*decision_tensor*, *override=False*)
Intercept the normal class for the given decision matrix, i.e. a normal class is assigned to each zero vector class assignment. E.g. the assignment *[0,0,0,0]* is transformed to *[1,0,0,0]*, under the assumption that *0* is a normal class.

Parameters

- **decision_tensor** – *numpy.array* of shape *(n_classifiers, n_samples, n_classes)*. Tensor of decision outputs by different classifiers per sample.
- **override** – If *true*, the class *0* is assumed as a normal class. Otherwise a new class is prepended to existing classes.

Returns *numpy.array* of shape *(n_samples, n_classes)* for *override=False*. *numpy.array* of shape *(n_samples, n_classes + 1)* for *override=True*. Matrix of decision outputs with intercepted normal class.

CHAPTER

FOUR

USAGE AND EXAMPLES

4.1 A simple example

The following code shows an illustrative and simple example of using pusion for decision outputs of three classifiers.

```
import pusion as p
import numpy as np

# Create exemplary classification outputs (class assignments)
classifier_a = [[0, 0, 1], [0, 0, 1], [0, 1, 0]]
classifier_b = [[0, 0, 1], [0, 1, 0], [1, 0, 0]]
classifier_c = [[0, 1, 0], [0, 1, 0], [0, 1, 0]]

# Create a numpy tensor
ensemble_out = np.array([classifier_a, classifier_b, classifier_c])

# Initialize the general framework interface
dp = p.DecisionProcessor(p.Configuration(method=p.Method.MACRO_MAJORITY_VOTE,
                                         problem=p.Problem.MULTI_CLASS,
                                         assignment_type=p.AssignmentType.CRISP,
                                         coverage_type=p.CoverageType.REDUNDANT))

# Fuse the ensemble classification outputs
fused_decisions = np.array(dp.combine(ensemble_out))

print(fused_decisions)
```

Output:

```
[[0 0 1]
 [0 1 0]
 [0 1 0]]
```

4.2 A richer example

In this example, an ensemble is created using *sklearn*'s neural network classifiers. The 200 classification outputs are split up into validation and test datasets. `y_ensemble_valid` and `y_ensemble_test` holds the classification outputs of the whole ensemble, while `y_valid` and `y_test` are representing true labels. The validation datasets are used to train the *DempsterShaferCombiner* combiner (*DS*), while the final fusion is performed on the test dataset (without true labels).

```
import pusion as p

import sklearn

# Create an ensemble of 3 neural networks with different hyperparameters
classifiers = [
    sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,)),
    sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100, 50)),
    sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100, 50, 25)),
]

# Generate samples for the ensemble
y_ensemble_valid, y_valid, y_ensemble_test, y_test = p.generate_multiclass_ensemble_
↪classification_outputs(
    classifiers=classifiers,
    n_classes=5,
    n_samples=200)

# User defined configuration
conf = p.Configuration(
    method=p.Method.DEMPSTER_SHAFER,
    problem=p.Problem.MULTI_CLASS,
    assignment_type=p.AssignmentType.CRISP,
    coverage_type=p.CoverageType.REDUNDANT
)

# Initialize the general framework interface
dp = p.DecisionProcessor(conf)

# Train the selected Dempster Shafer combiner with the validation dataset
dp.train(y_ensemble_valid, y_valid)

# Fuse the ensemble classification outputs (test dataset)
y_comb = dp.combine(y_ensemble_test)
```

4.3 Evaluation

In addition to the previous example, we are able to evaluate both, the ensemble and the combiner classification performance using the evaluation methods provided by the framework. The critical point for achieving a reasonable comparison is obviously the usage of the same test dataset for the combiner as well as for the ensemble.

```
# Define classification performance metrics used for the evaluation
eval_metrics = [
    p.PerformanceMetric.ACCURACY,
    p.PerformanceMetric.MICRO_F1_SCORE,
    p.PerformanceMetric.MICRO_PRECISION
]

print("============= Ensemble ===============")
eval_classifiers = p.Evaluation(*eval_metrics)
eval_classifiers.set_instances(classifiers)
eval_classifiers.evaluate(y_test, y_ensemble_test)
print(eval_classifiers.get_report())

print("============== Combiner ==============")
eval_combiner = p.Evaluation(*eval_metrics)
eval_combiner.set_instances(dp.get_combiner())
eval_combiner.evaluate(y_test, y_comb)
print(eval_combiner.get_report())
```

Output:

```
============= Ensemble ===============
                                    accuracy     f1  precision
MLPClassifier [0]                      0.810  0.810      0.810
MLPClassifier [1]                      0.800  0.800      0.800
MLPClassifier [2]                      0.792  0.792      0.792
============== Combiner ==============
                                    accuracy     f1  precision
DempsterShaferCombiner                 0.816  0.816      0.816
```

4.4 Auto Combiner

The following code shows an exemplary usage and evaluation of the *AutoCombiner* specified in the configuration.

```
dp = p.DecisionProcessor(p.Configuration(method=p.Method.AUTO))
dp.train(y_ensemble_valid, y_valid)
y_comb = dp.combine(y_ensemble_test)

eval_combiner = p.Evaluation(*eval_metrics)
eval_combiner.set_instances(dp.get_combiner())
eval_combiner.evaluate(y_test, y_comb)

dp.set_evaluation(eval_combiner)
print(dp.report())
```

Output:

```
================================== AutoCombiner - Report␣
↪==================================
                   Problem: MULTI_CLASS
           Assignment type: CRISP
             Coverage type: REDUNDANT
   Combiner type selection: UtilityBasedCombiner, TrainableCombiner
      Compatible combiners: CosineSimilarityCombiner, MacroMajorityVoteCombiner,␣
↪MicroMajorityVoteCombiner, SimpleAverageCombiner, BehaviourKnowledgeSpaceCombiner,␣
↪DecisionTemplatesCombiner, KNNCombiner, DempsterShaferCombiner,␣
↪MaximumLikelihoodCombiner, NaiveBayesCombiner, NeuralNetworkCombiner,␣
↪WeightedVotingCombiner
          Optimal combiner: CosineSimilarityCombiner
Classification performance:
                                    accuracy  micro_f1  micro_precision
AutoCombiner                           0.836     0.836            0.836
================================================================================================
```

4.5 Generic Combiner

For the given data sets one could also use the *GenericCombiner* to gain an overview over applicable methods and their respective performances.

```
dp = p.DecisionProcessor(p.Configuration(method=p.Method.GENERIC))
dp.train(y_ensemble_valid, y_valid)
dp.combine(y_ensemble_test)

eval_combiner = p.Evaluation(*eval_metrics)
eval_combiner.set_instances(dp.get_combiners())
eval_combiner.evaluate(y_test, dp.get_multi_combiner_decision_output())

dp.set_evaluation(eval_combiner)
print(dp.report())
```

Note: The *DecisionProcessor* provides `get_multi_combiner_decision_output()` to retrieve fused decisions from each applicable combiner.

Output:

```
================================= GenericCombiner - Report␣
↪==================================
                   Problem: MULTI_CLASS
           Assignment type: CRISP
             Coverage type: REDUNDANT
   Combiner type selection: UtilityBasedCombiner, TrainableCombiner
      Compatible combiners: CosineSimilarityCombiner, MacroMajorityVoteCombiner,␣
↪MicroMajorityVoteCombiner, SimpleAverageCombiner, BehaviourKnowledgeSpaceCombiner,␣
↪DecisionTemplatesCombiner, KNNCombiner, DempsterShaferCombiner,␣
↪MaximumLikelihoodCombiner, NaiveBayesCombiner, NeuralNetworkCombiner,␣
↪WeightedVotingCombiner
```

(continues on next page)

(continued from previous page)

```
          Optimal combiner: WeightedVotingCombiner
Classification performance:
                                    accuracy  micro_f1  micro_precision
CosineSimilarityCombiner               0.836     0.836            0.836
MacroMajorityVoteCombiner              0.836     0.836            0.836
MicroMajorityVoteCombiner              0.836     0.836            0.836
SimpleAverageCombiner                  0.836     0.836            0.836
BehaviourKnowledgeSpaceCombiner        0.822     0.831            0.840
DecisionTemplatesCombiner              0.836     0.836            0.836
KNNCombiner                            0.826     0.836            0.846
DempsterShaferCombiner                 0.836     0.836            0.836
MaximumLikelihoodCombiner              0.834     0.834            0.834
NaiveBayesCombiner                     0.836     0.836            0.836
NeuralNetworkCombiner                  0.826     0.832            0.838
WeightedVotingCombiner                 0.836     0.836            0.836
================================================================================================
```

4.6 CR classification

In *complementary-redundant* classification (CR), ensemble classifiers are not able to make predictions for all available classes. They may complement each other or share some classes. In such cases, a *coverage* needs to be specified in order to use the framework properly. The coverage describes for each ensemble classifier, which classes it is able to make predictions for. In pusion, it can be defined by a simple 2D list, e.g., `[[0,1], [0,2,3]]`, where the first classifier is covering the classes *0,1* while the second one covers *0,2,3*. The following code example shows how to generate and combine such complementary-redundant classification outputs.

```
import pusion as p
import sklearn

# Create an ensemble of 3 neural networks with different hyperparameters
classifiers = [
    sklearn.neural_network.MLPClassifier(max_iter=5000, hidden_layer_sizes=(100,)),
    sklearn.neural_network.MLPClassifier(max_iter=5000, hidden_layer_sizes=(100, 50)),
    sklearn.neural_network.MLPClassifier(max_iter=5000, hidden_layer_sizes=(100, 50,␣
↪25)),
]

# Create a random complementary-redundant classification coverage with 60% overlap.
coverage = p.generate_classification_coverage(n_classifiers=3, n_classes=5, overlap=.6,␣
↪normal_class=True)

# Generate samples for the complementary-redundant ensemble
y_ensemble_valid, y_valid, y_ensemble_test, y_test = p.generate_multilabel_cr_ensemble_
↪classification_outputs(
    classifiers=classifiers,
    n_classes=5,
    n_samples=2000,
    coverage=coverage)

# Initialize the general framework interface
```

(continues on next page)

(continued from previous page)

```
dp = p.DecisionProcessor(p.Configuration(method=p.Method.AUTO))

# Since we are dealing with a CR output, we need to propagate the coverage to the␣
↪`DecisionProcessor`.
dp.set_coverage(coverage)

# Train the AutoCombiner with the validation dataset
dp.train(y_ensemble_valid, y_valid)

# Fuse the ensemble classification outputs (test dataset)
y_comb = dp.combine(y_ensemble_test)
```

The framework provides also a specific evaluation methodology for complementary-redundant results.

```
# Define classification performance metrics used for the evaluation
eval_metrics = [
    p.PerformanceMetric.ACCURACY,
    p.PerformanceMetric.MICRO_F1_SCORE,
    p.PerformanceMetric.MICRO_PRECISION
]

# Evaluate ensemble classifiers
eval_classifiers = p.Evaluation(*eval_metrics)
eval_classifiers.set_instances("Ensemble")
eval_classifiers.evaluate_cr_decision_outputs(y_test, y_ensemble_test, coverage)
print(eval_classifiers.get_report())

# Evaluate the fusion
eval_combiner = p.Evaluation(*eval_metrics)
eval_combiner.set_instances(dp.get_combiner())
eval_combiner.evaluate_cr_decision_outputs(y_test, y_comb)

dp.set_evaluation(eval_combiner)
print(dp.report())
```

Output:

```
                                  accuracy  micro_f1  micro_precision
Ensemble                             0.804     0.804            0.804
================================= AutoCombiner - Report␣
↪=================================
                  Problem: MULTI_LABEL
           Assignment type: CRISP
             Coverage type: COMPLEMENTARY_REDUNDANT
   Combiner type selection: UtilityBasedCombiner, TrainableCombiner
      Compatible combiners: CRCosineSimilarity, CRMicroMajorityVoteCombiner,␣
↪CRSimpleAverageCombiner, CRDecisionTemplatesCombiner, CRKNNCombiner,␣
↪CRNeuralNetworkCombiner
          Optimal combiner: CRDecisionTemplatesCombiner
Classification performance:
                                  accuracy  micro_f1  micro_precision
AutoCombiner                         0.813     0.813            0.813
================================================================================================
```

Warning: Combiner output is always redundant, which means that all classes are covered for each sample. To make a reasonable comparison between the combiner and the ensemble use `evaluate_cr_*` methods for both.

CHAPTER

FIVE

LICENSE

MIT License

CHAPTER

SIX

ABOUT

This framework was developed in the context of a research project and several student works at the Institute of Parallel and Distributed Systems of the University of Stuttgart.

CHAPTER

SEVEN

INDICES AND TABLES

- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W